C/C++ Users Journal™
Advanced Solutions for Professional Developers

Building Hybrid Systems with Boost.Python

David Abrahams and Ralf W. Grosse-Kunstleve

Getting our two favorite languages to work together is so much easier now.

Introduction

Python and C++ are in many ways as different as two languages could be: while C++ is compiled to machine-code, Python is interpreted. Python's dynamic type system is cited as the foundation of its flexibility, while in C++ static typing is the cornerstone of its efficiency. C++ has an intricate compile-time meta-language, while in Python, practically everything happens at runtime.

Still, for many programmers, these very differences mean that Python and C++ complement one another perfectly. Performance bottlenecks in Python programs can be rewritten in C++ for maximal speed, and authors of powerful C++ libraries choose Python as a middleware language for its flexible system integration capabilities. Furthermore, the surface differences mask some strong similarities:

- 'C'-family control structures (**if**, **while**, **for**...)
- Support for object-orientation, functional programming, and generic programming (both are multi-paradigm programming languages.)
- Support for operator overloading
- High-level components such as collections and iterators.
- Encapsulation for reusable libraries (modules, namespaces)
- High-level error handling with exceptions
- C++ idioms in common use, such as handle/body classes and reference-counted smart pointers mirror Python reference semantics.

Given Python's rich 'C' API, it should in principle be possible to expose C++ interfaces to Python with an interface like that of their C++ counterparts. The facilities provided by Python alone for integration with C++ are meager, though: 'C' has only rudimentary abstraction facilities, and no support for exception-handling. Extensions are required to manage Python reference counts manually, which is both tedious and error-prone. They also tend to contain a great deal of boilerplate code repetition which makes them difficult to maintain, especially when wrapping an evolving C/C++ API.

These limitations have lead to the development of a variety of high-level wrapping systems, most of which introduce their own specialized languages to control the process. In contrast, Boost.Python presents the user with a high-level C++ interface for wrapping C++ classes and functions, managing complexity behind-the-scenes with static metaprogramming. Boost.Python also goes beyond the scope of earlier systems by providing:

- Support for overriding C++ virtual functions in Python.

- Comprehensive lifetime management facilities for low-level C++ pointers and references.
- Support for "component-based" independent development of interacting extensions
- Integration with Python's powerful serialization engine (pickle).
- Coherence with the C++ rules for type conversion.

The key insight that sparked the development of Boost.Python is that much of the boilerplate in traditional extension modules could be eliminated using the ability of C++ templates to "dissect" types. Each argument of a wrapped C++ function must be extracted from a Python object using a procedure that depends on the argument type. Similarly the function's return type determines how its result will be converted from C++ to Python. Of course, argument and return types are part of each function's type, so Boost.Python can deduce most of the information required from the types of (member) function pointers.

Development with Boost.Python is "user-guided": as much information is extracted directly from the source code to be wrapped as is possible within the limits of pure C++, and some additional information is supplied explicitly by the user. Mostly the process is mechanical and little intervention is required. Because the interface specification is written in the same full-featured language as the code being exposed, the user has unprecedented power available when she does need to take control.

Boost.Python Design Goals

The primary goal of Boost.Python is to allow users to expose C++ classes and functions to Python using nothing more than a C++ compiler. In broad strokes, the user experience should be one of directly manipulating C++ objects from Python.

However, it's also important not to translate all interfaces *too* literally: the idioms of each language must be respected. For example, though C++ and Python both have an iterator concept, the concepts are expressed very differently. Boost.Python has to be able to bridge the interface gap.

It must be possible to insulate Python users from crashes resulting from trivial misuses of C++ interfaces, such as accessing already-deleted objects. By the same token, the library should insulate C++ users from the low-level Python C API, replacing error-prone C interfaces like manual reference-count management and raw **PyObject** pointers with more-robust alternatives.

Support for component-based development is crucial so that C++ types exposed in one extension module can be passed to functions exposed in another extension without loss of crucial information like C++ inheritance relationships.

Finally, all wrapping must be *non-intrusive*. In other words, the wrapping must occur without modifying or even seeing the original C++ source code. Existing C++ libraries have to be wrappable by third parties who only have access to header files and binaries.

Hello Boost.Python World

And now for a preview of Boost.Python, and a description of how Boost.Python improves on the raw facilities offered by Python. Here's a function you might want to expose:

```
char const* greet(unsigned x)
{
   static char const* const msgs[] =
```

```
   { "hello", "Boost.Python", "world!" };

   if (x > 2)
       throw std::range_error("greet: index out of range");

   return msgs[x];
}
```

To wrap this function in standard C++ using the Python C API, you'd need something like this:

```
extern "C" // all Python interactions use 'C' linkage
           // and calling convention
{
    // Wrapper to handle argument/result conversion
    // and checking
    PyObject* greet_wrap(PyObject* args, PyObject * keywords)
    {
         int x;
         // extract/check arguments
         if (PyArg_ParseTuple(args, "i", &x))
         {
             // invoke wrapped function
             char const* result = greet(x);
             // convert result to Python
             return PyString_FromString(result);
         }
         // error occurred
         return 0;
    }

    // Table of wrapped functions to be exposed by the module
    static PyMethodDef methods[] = {
        { "greet", greet_wrap, METH_VARARGS,
            "return one of 3 parts of a greeting" }
        , { NULL, NULL, 0, NULL } // sentinel
    };

    // module initialization function
    DL_EXPORT init_hello()
    {
        // add the methods to the module
        (void) Py_InitModule("hello", methods);
    }
}
```

Now here's the wrapping code you'd use to expose it with Boost.Python:

```
#include <boost/python.hpp>
using namespace boost::python;
BOOST_PYTHON_MODULE(hello)
{
    def("greet", greet, "return one of 3 parts of a greeting");
}
```

and here is the code in action:

```
>>> import hello
>>> for x in range(3):
```

```
...     print hello.greet(x)
...
hello
Boost.Python
world!
```

Aside from the fact that the C API version is much more verbose than the Boost.Python version, it's worth noting that the C API doesn't handle a few things correctly:

- The original function accepts an unsigned integer, and the Python C API only gives us a way of extracting signed integers. The Boost.Python version will raise a Python exception if we try to pass a negative number to **hello.greet**, but the C API version will proceed to do whatever the C++ implementation does when converting a negative integer to unsigned (usually wrapping to some very large number) and pass the incorrect translation on to the wrapped function.
- That brings us to the second problem: if the C++ **greet()** function is called with a number greater than 2, it will throw an exception. Typically, if a C++ exception propagates across the boundary with code generated by a C compiler, it will cause a crash. As you can see in the first version, there's no C++ scaffolding to prevent this from happening. Functions wrapped by Boost.Python automatically include an exception-handling layer that protects Python users by translating unhandled C++ exceptions into a corresponding Python exception.
- A slightly more-subtle limitation is that the argument conversion used in the Python C API case can only get that integer **x** in one way. **PyArg_ParseTuple** can't convert Python **long** objects (arbitrary-precision integers) that happen to fit in an **unsigned int** but not in a **signed long**, nor will it ever handle a wrapped C++ class with a user-defined implicit **operator unsigned int()** conversion. The Boost.Python's dynamic type conversion registry allows users to add arbitrary conversion methods.

Library Overview

This section outlines some of the library's major features. Except as neccessary to avoid confusion, details of library implementation are omitted.

Exposing Classes

C++ **classes** and **struct**s are exposed with a similarly-terse interface.

Given:

```
struct World
{
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }
    std::string msg;
};
```

The following code will expose the preceding code in our extension module:

```
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(hello)
{
    class_<World>("World")
        .def("greet", &World::greet)
```

```
        .def("set", &World::set)
    ;
}
```

Although this code has a certain pythonic familiarity, people sometimes find the syntax bit confusing because it doesn't look like most of the C++ code they're used to. All the same, this is just standard C++. Because of their flexible syntax and operator overloading, C++ and Python are great for defining domain-specific (sub)languages (DSLs), and that's what we've done in Boost.Python. To break it down:

```
class_<World>("World")
```

constructs an unnamed object of type **class_<World>** and passes **"World"** to its constructor. This creates a new-style Python class called **World** in the extension module and associates it with the C++ type **World** in the Boost.Python type conversion registry. We might have also written:

```
class_<World> w("World");
```

but that would've been more verbose, since we'd have to name **w** again to invoke its **def()** member function:

```
    w.def("greet", &World::greet)
```

There's nothing special about the location of the dot for member access in the original example: C++ allows any amount of whitespace on either side of a token, and placing the dot at the beginning of each line allows us to chain as many successive calls to member functions as we like with a uniform syntax. The other key fact that allows chaining is that **class_<>** member functions all return a reference to ***this**.

So the example is equivalent to:

```
class_<World> w("World");
w.def("greet", &World::greet);
w.def("set", &World::set);
```

It's occasionally useful to break down the components of a Boost.Python class wrapper in this way, but the rest of this article will stick to the terse syntax.

For completeness, here's the wrapped class in use:

```
>>> import hello
>>> planet = hello.World()
>>> planet.set('howdy')
>>> planet.greet()
'howdy'
```

Constructors

Since our **World** class is just a plain **struct**, it has an implicit no-argument (default) constructor. Boost.Python exposes the nullary constructor by default, which is why we were able to write:

```
>>> planet = hello.World()
```

However, well-designed classes in any language may require constructor arguments in order to establish

their invariants. Unlike Python, where **__init__** is just a specially-named method, in C++, constructors cannot be handled like ordinary member functions. In particular, we can't take their address: **&World::World** is an error. The library provides a different interface for specifying constructors.

Given:

```
struct World
{
    World(std::string msg); // added constructor
    ...
```

We can modify our wrapping code as follows:

```
class_<World>("World", init<std::string>())
    ...
```

Of course, a C++ class may have additional constructors, and we can expose those as well by passing more instances of **init<...>** to **def()**:

```
class_<World>("World", init<std::string>())
    .def(init<double, double>())
    ...
```

Boost.Python allows you to overload wrapped functions, member functions, and constructors to mirror C++ overloading.

Data Members and Properties

Any publicly-accessible data members in a C++ class can be easily exposed as either **readonly** or **readwrite** attributes:

```
class_<World>("World", init<std::string>())
    .def_readonly("msg", &World::msg)
    ...
```

and can be used directly in Python:

```
>>> planet = hello.World('howdy')
>>> planet.msg
'howdy'
```

This does *not* result in adding attributes to the **World** instance **__dict__**, which can result in substantial memory savings when wrapping large data structures. In fact, no instance **__dict__** will be created at all unless attributes are explicitly added from Python. Boost.Python owes this capability to the new Python 2.2 type system, in particular, the descriptor interface and **property** type.

In C++, publicly-accessible data members are considered a sign of poor design because they break encapsulation, and style guides usually dictate the use of *getter* and *setter* functions instead. In Python, however, **__getattr__**, **__setattr__**, and since 2.2, **property** mean that attribute access is just one more well-encapsulated syntactic tool at the programmer's disposal. Boost.Python bridges this idiomatic gap by making Python **property** creation directly available to users. If **msg** were private, we could still expose it as an attribute in Python as follows:

```
class_<World>("World", init<std::string>())
    .add_property("msg", &World::greet, &World::set)
    ...
```

The example above mirrors the familiar usage of properties in Python 2.2+:

```
>>> class World(object):
...     __init__(self, msg):
...         self.__msg = msg
...     def greet(self):
...         return self.__msg
...     def set(self, msg):
...         self.__msg = msg
...     msg = property(greet, set)
```

Operator Overloading

The ability to write arithmetic operators for user-defined types has been a major factor in the success of both languages for numerical computation, and the success of packages like NumPy attests to the power of exposing operators in extension modules. Boost.Python provides a concise mechanism for wrapping operator overloads. The example below shows a fragment from a wrapper for the Boost rational number library:

```
class_<rational<int> >("rational_int")

  .def(init<int, int>()) // constructor, e.g. rational_int(3,4)

  .def("numerator", &rational<int>::numerator)

  .def("denominator", &rational<int>::denominator)

  .def(-self)        // __neg__ (unary minus)

  .def(self + self)  // __add__ (homogeneous)

  .def(self * self)  // __mul__

  .def(self + int()) // __add__ (heterogenous)

  .def(int() + self) // __radd__

  ...
```

The magic is performed using a simplified application of expression templates [1], a technique originally developed for optimization of high-performance matrix algebra expressions. The essence is that instead of performing the computation immediately, operators are overloaded to construct a type *representing* the computation. In matrix algebra, dramatic optimizations are often available when the structure of an entire expression can be taken into account, rather than evaluating each operation "greedily." Boost.Python uses the same technique to build an appropriate Python method object based on expressions involving **self**.

Inheritance

C++ inheritance relationships can be represented to Boost.Python by adding an optional **bases<...>**

argument to the **class_<...>** template parameter list as follows:

```
class_<Derived, bases<Base1,Base2> >("Derived")
     ...
```

This has two effects:

1. When the **class_<...>** is created, Python type objects corresponding to **Base1** and **Base2** are looked up in the Boost.Python registry and are used as bases for the new Python **Derived** type object, so methods exposed for the Python **Base1** and **Base2** types are automatically members of the **Derived** type. Because the registry is global, this works correctly even if **Derived** is exposed in a different module from either of its bases.

2. C++ conversions from **Derived** to its bases are added to the Boost.Python registry. Thus wrapped C++ methods expecting (a pointer or reference to) an object of either base type can be called with an object wrapping a **Derived** instance. Wrapped member functions of class **T** are treated as though they have an implicit first argument of **T&**, so these conversions are neccessary to allow the base class methods to be called for derived objects.

Of course it's possible to derive new Python classes from wrapped C++ class instances. Because Boost.Python uses the newstyle class system, derivation works very much as for the Python built-in types. There is, however, one significant difference: the built-in types generally establish their invariants in their **__new__** function, so that derived classes do not need to call **__init__** on the base class before invoking its methods:

```
>>> class L(list):
...       def __init__(self):
...           pass
...
>>> L().reverse()
>>>
```

Because C++ object construction is a one-step operation, C++ instance data cannot be constructed until the arguments are available in the **__init__** function:

```
>>> class D(SomeBPLClass):
...       def __init__(self):
...           pass
...
>>> D().some_bpl_method()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: bad argument type for built-in operation
```

This happened because Boost.Python couldn't find instance data of type **SomeBPLClass** within the **D** instance; **D**'s **__init__** function masked construction of the base class. The situation could be corrected by either removing **D**'s **__init__** function or having it call **SomeBPLClass.__init__(...)** explicitly.

Virtual Functions

Deriving new types in Python from extension classes is not very interesting unless the types can be used polymorphically from C++. In other words, Python method implementations should appear to override

the implementation of C++ virtual functions when called through base class pointers/references from C++. Since the only way to alter the behavior of a virtual function is to override it in a derived class, the user must build a special derived class to dispatch a polymorphic class's virtual functions:

```
//
// interface to wrap:
//
class Base
{
public:
    virtual int f(std::string x) { return 42; }
    virtual ~Base();
};

int calls_f(Base const& b, std::string x) { return b.f(x); }

//
// Wrapping Code
//

// Dispatcher class
struct BaseWrap : Base
{
    // Store a pointer to the Python object
    BaseWrap(PyObject* self_) : self(self_) {}
    PyObject* self;

    // Default implementation, for when f is not overridden
    int f_default(std::string x) { return this->Base::f(x); }
    // Dispatch implementation
    int f(std::string x)
    { return call_method<int>(self, "f", x); }
};

...
    def("calls_f", calls_f);
    class_<Base, BaseWrap>("Base")
        .def("f", &Base::f, &BaseWrap::f_default)
        ;
```

Now here's some Python code that demonstrates:

```
>>> class Derived(Base):
...     def f(self, s):
...          return len(s)
...
>>> calls_f(Base(), 'foo')
42
>>> calls_f(Derived(), 'forty-two')
9
```

Things to notice about the dispatcher class:

- The key element that allows overriding in Python is the **call_method** invocation, which uses the same global type conversion registry as the C++ function wrapping does to convert its arguments from C++ to Python and its return type from Python to C++.
- Any constructor signatures you wish to wrap must be replicated with an initial **PyObject***

argument.
- The dispatcher must store this argument so that it can be used to invoke **call_method**.
- The **f_default** member function is needed when the function being exposed is not pure virtual; there's no other way **Base::f** can be called on an object of type **BaseWrap**, since it overrides **f**.

Admittedly, this formula is tedious to repeat, especially on a project with many polymorphic classes; that it is neccessary reflects a limitation of C++: there's no way to enumerate the members of a class and find out which are virtual functions.Thankfully, Bruno da Silva de Oliveira has contributed a front-end to Boost.Python which uses GCC to parse headers and generate wrapping code automatically, driven by a very small amount of user-written Python. His system, called "Pyste", is becoming very popular, and shows that there are advantages to hybrid development for Boost.Python itself.

Object Interface

Experienced C language extension module authors will be familiar with the ubiquitous **PyObject***, manual reference-counting, and the need to remember which API calls return "new" (owned) references or "borrowed" (raw) references. These constraints are not just cumbersome but are also a major source of errors, especially in the presence of exceptions.

Boost.Python provides a class **object** that automates reference counting and provides conversion to Python from C++ objects of arbitrary type. This feature significantly reduces the learning effort for prospective extension module writers.

Creating an **object** from any other type is extremely simple:

```
object s("hello, world");  // s manages a Python string
```

object has templated interactions with all other types, with automatic to-Python conversions. It happens so naturally that it's easily overlooked:

```
object ten_Os = 10 * s[4]; // -> "oooooooooo"
```

In the example above, **4** and **10** are converted to Python objects before the indexing and multiplication operations are invoked.

The **extract<T>** class template can be used to convert Python objects to C++ types:

```
double x = extract<double>(o);
```

If a conversion in either direction cannot be performed, an appropriate exception is thrown at runtime.

The **object** type is accompanied by a set of derived types that mirror the Python built-in types such as **list**, **dict**, **tuple**, etc. as much as possible. This enables convenient manipulation of these high-level types from C++:

```
dict d;
d["some"] = "thing";
d["lucky_number"] = 13;
list l = d.keys();
```

This almost looks and works like regular Python code, but it is pure C++. Of course we can also wrap

C++ functions that accept or return **object** instances.

Thinking hybrid

Because of the practical and mental difficulties of combining programming languages, it is common to settle on a single language at the outset of any development effort. For many applications, performance considerations dictate the use of a compiled language for the core algorithms. Unfortunately, due to the complexity of the static type system, the price we pay for runtime performance is often a significant increase in development time. Experience shows that writing maintainable C++ code usually takes longer and requires far more hard-earned working experience than developing comparable Python code. Even when developers are comfortable working exclusively in compiled languages, they often augment their systems by some type of ad hoc scripting layer for the benefit of their users without ever availing themselves of the same advantages.

Boost.Python enables us to "think hybrid." Python can be used for rapidly prototyping a new application; its ease of use and the large pool of standard libraries give us a head start on the way to a working system. If necessary, the working code can be used to discover rate-limiting hotspots. To maximize performance these hotspots can be reimplemented in C++, together with the Boost.Python bindings needed to tie them back into the existing higher-level procedure.

Of course, this top-down approach is less attractive if it is clear from the start that many algorithms will eventually have to be implemented in C++. Fortunately Boost.Python also enables us to pursue a bottom-up approach. We have used this approach very successfully in the development of a toolbox for scientific applications. The toolbox started out mainly as a library of C++ classes with Boost.Python bindings, and for a while the growth was mainly concentrated on the C++ parts. However, as the toolbox is becoming more complete, more and more newly added functionality can be implemented in Python.

Figure 1 shows the estimated ratio of newly added C++ and Python code over time as new algorithms are implemented. We expect this ratio to level out near 70% Python. Being able to solve new problems mostly in Python rather than a more difficult statically typed language is the return on our investment in Boost.Python. The ability to access all of our code from Python allows a broader group of developers to use Python in the rapid development of new applications.

Conclusions

Boost.Python achieves seamless interoperability between two rich and complementary language environments. Because Boost.Python leverages template metaprogramming to introspect about types and functions, the user never has to learn a third syntax: the interface definitions are written in concise and maintainable C++. Also, the wrapping system doesn't have to parse C++ headers or represent the type system: the compiler does that work for us.

Computationally intensive tasks play to the strengths of C++ and are often impossible to implement efficiently in pure Python, while jobs like serialization that are trivial in Python can be very difficult in pure C++. Given the luxury of building a hybrid software system from the ground up, we can approach design with new confidence and power.

Citations

[1] T. Veldhuizen, "Expression Templates," C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31. http://osl.iu.edu/~tveldhui/papers/ Expression-Templates/exprtmpl.html.

About the Authors

Dave Abrahams is a founding member of Boost.org and an active participant in ANSI/ISO C++ standardization. His experience in the software industry include shrink-wrap software development, embedded systems design and natural language processing. He is an author of six Boost libraries, and has made contributions to numerous others. Dave's other pursuits include bicycle racing, songwriting, and drawing. Since 2001 his company, Boost Consulting, has provided support, development, and training services focused on the Boost libraries. You can reach him through the Boost Consulting website at www.boost-consulting.com.

Ralf Grosse-Kunstleve is a scientist in the Physical Biosciences Division of the Lawrence Berkeley National Laboratory in California. He is part of the Computational Crystallography Initiative (cci.lbl.gov) which is leading an international effort aimed at advancing the automation of protein structure determination. The software system being developed by the collaboration has evolved together with the Boost.Python library and was designed from its inception as a hybrid Python/C++ system. The core components are available as an open source toolbox at cctbx.sourceforge.net.